In this lesson we are looking at indexes. Indexes allow our table to perform searches a lot faster.

SELECT \* FROM users;

|  |  |
| --- | --- |
| id  integer | name  character varying |
| 1 | Mary |
| 2 | Rolf |
| 3 | Jose |

For example, say we have got this table, but instead of just having three entries we have thousands of entries in our table. If we want to select a user searching by the name and not by the id. We need to set our query as below,

SELECT \* FROM users WHERE name= ‘Rolf’;

|  |  |
| --- | --- |
| id  integer | name  character varying (100) |
| 2 | Rolf |

This did not take us a long time to get this result from Postgres, but if we had thousands of users then to perform a search like this would have taken a long time. This is because the *name* is just a normal column, the id is a PRIMARY KEY and that makes it special, but name is just a normal column that has some data.

This is kind of like a dictionary where think of the id as the word in the dictionary and the name as the definition of the word. We would never search a dictionary by its definition, we would always search by the word because the words tend to be alphabetically ordered and therefore it is easy to search. The query that we ran above is like telling Postgres to search by the definition and naturally the only way to do that is to go every row in the table until it finds one that matches.

SELECT \* FROM users WHERE name= ‘Rolf’ LIMIT 1;

|  |  |
| --- | --- |
| id  integer | name  character varying (100) |
| 2 | Rolf |

If we have LIMIT 1, then that is a lot better, because as soon as Postgres finds one row with the name ‘Rolf’ it stops. But if we do not have LIMIT 1, Postgres needs to go absolutely every row of our table. That process is not very fast.

What we can do is allow Postgres to know that the names are ordered in a way that it can search nicely. We are going to order the names kind of alphabetically, though it is not exactly like that, but we can think of it like that. We are going to order the names in a binary tree.

An index orders the data in such a way that allows Postgres to search in a lot faster and in a lot more efficiently. That’s what an INDEX is for. Its just like an INDEX in a dictionary that has all the words, the INDEX allows us to search faster and more efficiently.

However, an Index is not free. It takes disc space because we have to store that Binary Tree somewhere and it is slower to insert data, to update data because then the Binary Tree that holds that structure that allows us to search has to be updated whenever we perform an insert or an update in our table.

That is why we should not be creating indexes for things we are not going to search by and if possible do not create indexes and just search by ID but most cases that is not possible.

For example, if we had a table with weather information each row might have an ID, but we need to search by location, so it would make sense to give the location an index as well. We might not search by temperature, so it would not make sense to give the temperature an index.

The PRIMARY KEY is an index by default, and it is an unique index, it is an special type of index, we are going to look at that.

Let’s create an index for our name’s column so we can search by names faster.

CREATE INDEX users\_name\_index

ON users(name);

Our index is set up, now if we make a search again.

SELECT \* FROM users

WHERE name= “Rolf”;

Now, if we make a search again its not going to be much faster.

SELECT \* FROM users

WHERE name= “Rolf”;

|  |  |
| --- | --- |
| id  integer | name  character varying (100) |
| 2 | Rolf |

That is because there are very few rows and that does not take much time at all. It would have been faster if we had lot of data.

The ID is a unique index, because each of the item on the binary tree are unique because the id is a unique column. The name column is not unique but if we had another column that is unique other than the id, then we could have created another unique index and the syntax is as below,

CREATE UNIQUE index\_name

ON table(column\_name);

We can also create a multiple column indexes which are useful only in one scenario. We should not create multi-column indexes arbitrarily. For example, we were performing our search depending on two columns as below,

SELECT \* FROM movies

WHERE id=1

AND user\_id=2;

We are not performing on *user\_id* or *id* alone, but we are performing on both columns simultaneously. Its more efficient to set up a multi-column index if we are doing and filtering.

The way we perform a search on a multi-column index is as below,

CREATE INDEX index\_name

ON movies(id, user\_id);

That’s how we write a query for multi-column indexes and that is only useful when we are doing and filtering on our data. If we are filtering data by the *user\_id* only or by *id* only then we do not need a multi-column index. Just have an index on *id* and another index on the *user\_id.*

Indexes can get corrupted in some scenarios and when they do become corrupted then our index search would not work and all our searches will be a lot slower and we can fix that by doing REINDEX. We can REINDEX a specific index and the REINDEX will delete that index and build a new index and that takes a bit of time but it will fix any index problem that we have got and in some scenarios if we are performing a lot of UPDATE or INSERTs the index can become really big and occupy a lot of space and it helps to do this if we are doing a lot od UPDATEs.

REINDEX INDEX users\_name\_index;

We can also REINDEX a table that does REINDEX for all the indexes in the table and, we can REINDEX a database the REINDEX all the indexes in our database. This is useful for indexes in our database when the indexes that the database gets corrupted. The database depends on several indexes those are called system indexes and if they get corrupted, our database will not start-up.

On that case we can tell Postgres to not use any indexes and then our database will start-up and the we can REINDEX our database and that will take some time and then we can restart Postgres telling it to use indexes.